Computation of normal form coefficients of cycle bifurcations of maps by algorithmic differentiation

J.D. Pryce[a], R. Khoshsiar Ghaziani[b], V. De Witte[c,*], W. Govaerts[c]

[a]*Department of Information Systems, Cranfield University, Shrivenham Campus SN6 8LA, UK*
[b]*Department of Mathematics, Shahrekord University, P.O. Box 115, Shahrekord, Iran*
[c]*Department of Applied Mathematics and Computer Science, Ghent University, Krijgslaan 281-S9, B-9000 Gent, Belgium*

Abstract

As an alternative to symbolic differentiation (SD) and finite differences (FD) for computing partial derivatives, we have implemented algorithmic differentiation (AD) techniques into the MATLAB bifurcation software CL_MATCONTM, `http://sourceforge.net/projects/matcont`, where we need to compute derivatives of an iterated map, with respect to state variables. We use derivatives up to the fifth order, of the iteration of a map to arbitrary order. The multilinear forms are needed to compute the normal form coefficients of codimension-1 and -2 bifurcation points. Methods based on finite differences are inaccurate for such computations.

Computation of the normal form coefficients confirms that AD is as accurate as SD. Moreover, elapsed time in computations using AD grows linearly with the iteration number J, but more like J^d for dth derivatives with SD. For small J, SD is still faster than AD.

Key words: Bifurcation, multilinear form, Taylor series, Matlab, iterated map

1. Introduction and motivation

In this paper we consider the iterated map

$$f^{(J)}(x,\alpha) = \underbrace{f(f(f(\cdots f}_{J \text{ times}}(x,\alpha)\cdots,\alpha),\alpha),\alpha). \tag{1}$$

where $x \in \mathbb{R}^n$ is a vector of state variables, $\alpha \in \mathbb{R}^p$ is a vector of parameters and $f : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^n$ is a nonlinear map. A J-cycle is a J-tuple $(x_1, ..., x_J)$ for which $f(x_i) = x_{i+1}$ $(i = 1, ..., J-1)$ and $f(x_J) = x_1$.

We discuss our experience in using algorithmic differentiation techniques (sometimes called automatic differentiation) as an aid in computing the numerical continuation and bifurcation of cycles. In particular, we consider the computation of the multilinear forms via the Taylor expansion, up to the fifth order, of an iterated map.

*Corresponding author
Email addresses: `j.d.pryce@ntlworld.com` (J.D. Pryce), `khoshsiar@sci.sku.ac.ir` (R. Khoshsiar Ghaziani), `Virginie.DeWitte@UGent.be` (V. De Witte), `Willy.Govaerts@UGent.be` (W. Govaerts)

Derivatives of first or second order can be approximated by finite difference methods reasonably well if sophisticated heuristics are used; for higher order derivatives this is a hopeless task. For our application, owing to the need for highly accurate derivatives of order up to five, one is forced to turn to algorithmic differentiation or symbolic approaches.

Many users of Matlab do not have the Matlab Symbolic Toolbox. Therefore, in our continuation and bifurcation software [7] we provide the option for AD. As we will show in the paper, AD is useful even if the Symbolic Toolbox is available because AD is faster for high values of the iteration number J. High values of J are often needed because cascades of period doubling bifurcations are generic in all maps that depend on at least one parameter. In practice this means that if a map from a finite-dimensional real space into itself is sufficiently smooth, and if it depends on a parameter, then it is likely that the map has J-cycles with an arbitrarily high J, for suitable parameter values. In standard texts this phenomenon is often used as an introduction to chaos.

We emphasize that in the computation of multilinear forms we do not need the tensors of partial derivatives, e.g the Hessians, and only derivatives with respect to one scalar variable are computed. This is sufficient to compute the directional derivatives that we need. However, the computation of normal form coefficients requires in many cases the solution of linear systems with the Jacobian matrix of $f^{(J)}$ or a matrix related to this Jacobian. Each column of the Jacobian of $f^{(J)}$ is then computed separately, and the $i-$th column is the one-linear form of $f^{(J)}$, applied to the $i-$th unit vector. The cost of this application is small compared to the cost of the high-order multilinear forms.

This paper is outlined as follows. Section 2 gives a brief background on AD followed by a discussion of techniques to compute derivatives of a given function. Section 3 presents a detailed description of the use of AD to compute multilinear forms that arise when finding the normal form coefficients of bifurcations of codimensions 1 and 2 (referred to as *codim-1 and codim-2 bifurcations* henceforth). Section 4 presents some numerical results and a comparison of time complexity using AD and SD in our application. Section 5 gives some conclusions and remarks.

2. Algorithmic differentiation background

2.1. Theory

To compute derivatives we can use algorithmic differentiation (AD). Every function f expressible by a computer program is built from a finite set of elementary functions (this term includes the basic arithmetic operations). The basic principle of AD, see Griewank [9] and Griewank and Walther [10], is to use the known formulae for differentiating elementary functions, together with the chain rule, to build up the needed derivatives of an arbitrary f.

We assume f is a vector function $y = f(x)$ over the reals with n real *inputs*, or independent variables, $x = (x_1, \ldots, x_n)$ and m real *outputs* $y = (y_1, \ldots, y_m)$. The code for f may contain branches and loops. However, each evaluation of f at given inputs x can be written as a *code list*, which is a finite sequence of assignments of the simple form

$$v_i = e_i(\text{previously defined } v_j\text{'s, or constants}), \quad i = 1, 2, \ldots, p+m, \tag{2}$$

where each e_i is one of the elementary functions. The v_i are called *variables*. In (2) it is convenient to use $v_{1-n}, \ldots, v_0$ as aliases for the inputs $x_1, \ldots, x_n$ and $v_{p+1}, \ldots, v_{p+m}$ as aliases for the outputs $y_1, \ldots, y_m$, following the notation of [9]. The remaining variables $v_1, \ldots, v_p$ are called *intermediate*.

The *forward mode* of AD is the simplest, and is appropriate to our application. Each v_i is represented by an object $\mathtt{v}_i$ of a data type that holds not just the value but some needed set of derivatives. For our purposes:

- There is (at any one point) just one variable being treated as independent: we call it t. Thus each variable v_i is regarded as a function of t.
- The data structure holds *Taylor coefficients* (TCs), that is the coefficients of the truncated Taylor series of v_i, up to some order p, expanded about some point $t = a$. Changing to a new independent variable $s = t - a$:

$$\mathtt{v}_i \text{ holds } (v_{i,0}, v_{i,1}, \ldots v_{i,p}) \quad \text{where} \quad v(a+s) = v_{i,0} + v_{i,1}s + \cdots + v_{i,p}s^p + O(s^{p+1}).$$

We call the data type `adtayl`. It is helpful to think of an `adtayl` object as representing an *infinite* power series which is only known up to the order p term. The Taylor coefficients are of course just scaled derivatives, and are simpler to manipulate than are derivatives.

When evaluating a code list, each elementary operation on real arguments is replaced by the corresponding operation on `adtayl` arguments regarded as power series known up to a certain order. Consider first the four basic arithmetic operations. Agree that `a` holds $(a_0, a_1 \ldots)$, and so on for other named variables. Let `a` be defined to order p, and `b` to order q. Then $\mathtt{c} = \mathtt{a} + \mathtt{b}$ and $\mathtt{d} = \mathtt{a} \times \mathtt{b}$ are defined to order $r = \min(p, q)$ by

$$\begin{aligned} c_i &= a_i + b_i, \\ d_i &= a_0 b_i + a_1 b_{i-1} + \cdots + a_i b_0, \end{aligned}$$

and similarly for $\mathtt{a} - \mathtt{b}$, and for $\mathtt{a} \div \mathtt{b}$ provided $b_0 \neq 0$.

For example, suppose $y = (2 + t)(3 + t^2)$ and we wish to obtain the power series of y up to order $p = 2$, expanded about the point $t = 1$. We initialise the process by creating the object representing the independent variable expanded to order 2 in terms of $s = t - 1$:

$$\mathtt{t} = (t_0, t_1, t_2) = (1, 1, 0) \text{ representing } 1 + 1s + 0s^2.$$

We create objects `c2` and `c3` representing the constant functions 2 and 3 respectively. The whole computation is shown in the following table.

Computation	Holds	Represents
`t = indep`	$(1, 1, 0)$	$t = 1 + s$
`c2 = const`	$(2, 0, 0)$	2
`c3 = const`	$(3, 0, 0)$	3
`v1 = c2 + t`	$(3, 1, 0)$	$2 + t = 3 + s$
`v2 = t * t`	$(1, 2, 1)$	$t^2 = 1 + 2s + s^2$
`v3 = c3 + v2`	$(4, 2, 1)$	$3 + t^2 = 4 + 2s + s^2$
`outputy = v1 * v3`	$(12, 10, 5)$	$(2 + t)(3 + t^2) = 12 + 10s + 5s^2 + O(s^3)$

For applying the standard functions $\exp, \cos, \ldots$ to power series there are various formulas in the literature. We have aimed to choose ones that can be made reasonably fast in MATLAB, especially when the argument is a vector of power series, not just a single one. This is not the place for details but we give a few examples.

First, multiplication of power series is a convolution, which can be realised by the very fast built-in `filter` function of MATLAB.

Second, the method for exp(a) exploits the fact that if $c(t) = \exp(a(t))$ then $c'(t) = a'(t)c(t)$, which is converted to the integral form $c(t) = c_0 + \int_0^t a'(s)c(s)\,\mathrm{d}s$ where $c_0 = e^{a_0}$. In terms of the coefficients this reduces to a triangular linear system, which is fast in MATLAB.

Third, for sin(a) and cos(a), it is convenient to compute both simultaneously as the real and imaginary parts of exp(i a), and to record both results, along with the argument a, in persistent storage. Each time a new a is given, it is checked against the recorded one. If they are equal, the result can be returned at once. Since cos and sin at the same argument often occur together in applications, this reduces the cost for these functions by nearly half.

Regarding previous work to provide AD facilities in MATLAB, Rich and Hill [15] provided a limited facility that enabled AD of simple expressions defined by a character string. The first significant work was that of Coleman and Verma [2, 3], who produced an operator-overloading AD package ADMAT with facilities for forward and reverse mode AD for first and second derivatives, and run-time Jacobian sparsity detection. These authors interfaced ADMAT with ADMIT [4], a package for efficient sparse Jacobian calculation, and recently the ADiMat hybrid source-transformation/operator-overloading AD tool [16] has been developed. For first derivatives, currently the most efficient and comprehensive tool is probably Forth's MAD package [8]. However, none of these tools handles Taylor series.

2.2. Data structure and interface of the MATLAB `adtayl` *class objects*

The `adtayl` data type was implemented as a MATLAB class of the same name. An `adtayl` object `x` has one field `tc`. Here `x` can be a scalar, a vector or a matrix. In the scalar case, `tc` is a row vector of length $(p + 1)$ holding the TCs $x_0, x_1, \ldots x_p$ of a variable $x = x(t)$ around a point $t = a$. MATLAB arrays are numbered from 1, so x_r is in position $r+1$ of `tc` for each r. In general, `tc` holds an $m \times n \times (p+1)$ array with the obvious meaning, with the TCs always along the third dimension. Thus $m = n = 1$ for the scalar case, and $m = 1$ or $n = 1$ for a row vector or column vector respectively.

One cannot create a general series (1) directly. `adtayl` creates the truncated Taylor series of the independent variable t, that is

$$t = a + 1s + 0s^2 + \ldots + 0s^p.$$

The MATLAB command for this is

```
t = adtayl(a, p);
```
(3)

One can create constant-functions, after creating the independent variable. Suppose `cval` holds the numeric value c. Then

```
c = adtayl(cval);
```
(4)

sets `c` to the `adtayl` object representing $c + 0s + \ldots + 0s^p$, where p comes from the current (most recently created) independent variable. `cval` can be a scalar or a one or two dimensional array, creating an `adtayl` object of the same shape. Such named constants are not needed often, as real constants in arithmetic expressions are converted automatically. All other functions must be calculated from (3) and (4).

For instance, the calculation of the Taylor expansion of $(2 + t)(3 + t^2)$ around $t = 1$ and up to order 2, discussed in subsection 2.1, can be done by the MATLAB statements

```
>>  t = adtayl(1,2);
>>  (2+t)*(3+t*t)
```

and the result will be displayed in the command window:

```
 Coefficients of orders 0 to 2 are:
    12    10     5
>>
```

2.3. Handling MATLAB `adtayl` *class objects*

All the operations are element-wise, so multiplication and division are the MATLAB operations `.*` and `./` (there is no `.\`). Matrix operations are not supported at all: the matrix multiplication operator `*` only works for scalar objects, being just a case of `.*`, and similarly for matrix division, power etc.

The following MATLAB standard functions are implemented for the `adtayl` class:

> `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `cotan`, `sec`, `csec`, `asin`, `acos`, `atan`, `acotan`, `asec`, `acsec`, `sinh`, `cosh`, `tanh`, `cotanh`, `sech`, `csech`, `asinh`, `acosh`, `atanh`, `acotanh`, `asech`, `acsech`.

The input and output of each function is a scalar, vector or matrix array of Taylor coefficients, input and output having the same dimensions. The handling of vector/matrix arguments is so far not always efficient. For the applications to date, only scalar values were needed. User-programmed functions should be written in terms of the above standard functions. However, the following MATLAB housekeeping functions are implemented for vector/matrix arguments.

- `display` prints an object in the command window.
- The class supports standard MATLAB array subscripting for referencing (`subsref`) and assigning (`subsasgn`) elements, or sections, of arrays, and assembling arrays using MATLAB's square bracket notation (`horzcat`, `vertcat`). E.g. `[t y; 1+t y*y]` creates a 2 by 2 matrix of Taylor series. Higher-dimensional arrays are not supported, and subscripting applies to the *m* and *n* directions only.
- Functions `size`, `numel`, and `end` are overloaded to give the correct behaviour of array accesses.
- By design, one cannot access the TC dimension with the above functions. There is an `order` function that returns the order *p* of an object; and a `tcs` function that extracts its TCs as an $[m, n, p+1]$ array. If *m* or *n* is 1, the singleton dimension is "squeezed" out to give a normal 2D array, which is easier to manipulate and display.

3. Computing multilinear forms

Bifurcation theory of maps (discrete dynamical systems) relies upon coordinate transformations to study qualitative properties of maps. These coordinate transformations depend upon derivatives of the maps. Thus, algorithmic differentiation provides an attractive technology for numerically studying bifurcations of dynamical systems.
In this section, we describe how to use the `adtayl` class to calculate the multilinear forms that

arise in the normal form coefficients of codim-1 and codim-2 bifurcation of cycles. The sign and size of these coefficients determine the bifurcation scenario near a local bifurcation point, cf. [11, 13, 14]. First we introduce the multilinear forms $A^{(J)}(q_1)$, $B^{(J)}(q_1,q_2)$, $C^{(J)}(q_1,q_2,q_3)$, $D^{(J)}(q_1,q_2,q_3,q_4)$ and $E^{(J)}(q_1,q_2,q_3,q_4,q_5)$ of order 1, 2, 3, 4 and 5, respectively. Assuming sufficient smoothness of a given function f in (1), we write

$$\begin{aligned} f^{(J)}(x_0+u,\alpha_0) &= x_0 + A^{(J)}(u) + \tfrac{1}{2}\,B^{(J)}(u,u) + \tfrac{1}{6}\,C^{(J)}(u,u,u) \\ &\quad + \tfrac{1}{24}\,D^{(J)}(u,u,u,u) + \tfrac{1}{120}\,E^{(J)}(u,u,u,u,u) + O(\|u\|^6), \end{aligned} \tag{5}$$

where the actions of the multilinear functions $A^{(J)}, B^{(J)}, C^{(J)}, D^{(J)}$, and $E^{(J)}$ are given by

$$\begin{aligned} A^{(J)}(x) &= \sum_{j=1}^{n} \frac{\partial f^{(J)}(x_0,\alpha_0)}{\partial \xi_j} x_j, \quad \text{(same as multiplication by } (f^{(J)})_x(x_0)) \\ B^{(J)}(x,y) &= \sum_{j,k=1}^{n} \frac{\partial^2 f^{(J)}(x_0,\alpha_0)}{\partial \xi_j \partial \xi_k} x_j y_k, \\ C^{(J)}(x,y,z) &= \sum_{j,k,l=1}^{n} \frac{\partial^3 f^{(J)}(x_0,\alpha_0)}{\partial \xi_j \partial \xi_k \partial \xi_l} x_j y_k z_l, \\ D^{(J)}(x,y,z,u) &= \sum_{j,k,l,m=1}^{n} \frac{\partial^4 f^{(J)}(x_0,\alpha_0)}{\partial \xi_j \partial \xi_k \partial \xi_l \partial \xi_m} x_j y_k z_l u_m, \\ E^{(J)}(x,y,z,u,v) &= \sum_{j,k,l,m,s=1}^{n} \frac{\partial^5 f^{(J)}(x_0,\alpha_0)}{\partial \xi_j \partial \xi_k \partial \xi_l \partial \xi_m \partial \xi_s} x_j y_k z_l u_m v_s, \end{aligned}$$

for $J = 1, 2, \ldots$

3.1. Computing the forms by AD

These multilinear forms can be computed by using the `adtayl` class and computing directional derivatives that are stored as an array of class `double`. We first define a function that iterates the map f a desired number of times. Here the argument `func` is (the function-handle of) the map f.

```
function y1 = Tmap(func,x0,hc,par,taylorder,J)
  s = adtayl(0,taylorder); %Base point & Taylor order
  y1= x0 + s*hc;
  for i=1:J
    y1 = func(0, y1, par{:});
  end
```

We now give the code for `multilinear1AD` and `multilinear2AD`, which compute $A^{(J)}(q_1)$ and $B^{(J)}(q_1,q_2)$ respectively, using `Tmap`. In their input lists, `q1` and `q2` are the n-vectors q_1, q_2, where n is the dimension of the phase space of the map. `x0` and `par` are the vector of state variables and of parameter values respectively, at the bifurcation point. `J` is the iteration number for the map. Similar code can be used for the higher-order multilinear forms.

```
function ytayl1 = multilinear1AD(func,q1,x0,par,J)
  taylorder = 1;
  y1 = Tmap(func,x0,q1,par,taylorder,J);
```

```
  ytayl1 = tcs(y1);

function ytayl2 = multilinear2AD(func,q1,q2,x0,par,J)
  taylorder = 2;
  if q1==q2
    y1 = Tmap(func,x0,q1,par,taylorder,J);
  else
    y11 = Tmap(func,x0,q1+q2,par,taylorder,J);
    y12 = Tmap(func,x0,q1-q2,par,taylorder,J);
    y1 = 1/4.0*(y11-y12);
  end
  ytayl2 = tcs(y1);
```

At the end, $A^{(J)}(q_1)$ is the last column of `ytayl1`, that is `ytayl1(:,end)`; and $B^{(J)}(q_1,q_2)$ is twice the last column of `ytayl2`, namely `2*ytayl2(:,end)`.

In the definition of `ytayl2` we used the polarization identity

$$B(u,v) = \frac{1}{4}(B(u+v,u+v) - B(u-v,u-v)),$$

where B is any bilinear form. Similar identities exist for the higher order forms, see [13], §10.3.4.

It only remains to provide code for a specific map f. The "Cod Stock"model in Test case 2 in §4 may be coded as follows:

```
function y = CodStockFunc(t,x,F,P,beta1,beta2,beta3,mu1,mu2)
  x1 = x(1); x2 = x(2);
  y = [ F*exp(-beta1*x2)*x2 + (1-mu1)*exp(-beta2*x2)*x1;
        P*exp(-beta3*x2)*x1 + (1-mu2)*x2 ];
```

To evaluate $A^{(J)}(q_1)$ at specific q_1 and q_2 one can type the following at the MATLAB command line. Note that `q1` and `x0` are ordinary column vectors, while `par` is a MATLAB cell-array, which is transformed into part of the list of arguments to the map; it holds $F, P, \beta_1, \beta_2, \beta_3, \mu_1, \mu_2$ in that order. `@CodStockFunc` is the function-handle for the `CodStockFunc` function.

```
>> par={399.5681,0.5, 1,1,1, 0.5,0.444715}
>> x0=[26.0; 3.0]
>> q1=[1;2]
>> q2=[3;4]
>> J=15
>> ytayl1=multilinear1AD(@CodStockFunc,q1,x0,par,J)
>> ytayl2=multilinear2AD(@CodStockFunc,q1,q2,x0,par,J)
>> A=ytayl1(:,end)
>> B=2*ytayl2(:,end)
```

When the above was run it produced these results:

```
ytayl1 =
   2.3896e+01   7.7278e+01
   3.0477e+00  -3.1623e-01
ytayl2 =
            0   7.7688e+01   1.9340e+03
            0  -3.1796e-01   7.4153e-01
A =
```

```
   7.7278e+01
  -3.1623e-01
B =
   3.8680e+03
   1.4831e+00
>>
```

3.2. Comparison with symbolic derivatives

If the symbolic toolbox of MATLAB is available then we can compute derivatives of (1), using recursive formulas, see [11]. Here follows a brief description of the recursive formulas, since we need to refer to them. The iteration of (1) gives rise to a sequence of points

$$\{x^1, x^2, x^3, \ldots, x^{K+1}\},$$

where $x^{J+1} = f^{(J)}(x^1, \alpha)$ for $J = 1, 2, \ldots, K$. Suppose that symbolic derivatives of f up to order 5 can be computed at each point. What follows is a straightforward application of the Chain Rule.

$$A^{(J)}q = A(x^K)A(x^{K-1})\cdots A(x^1)q, \tag{6}$$

$$B^{(J)}(q_1, q_2) = B(x^J)(A^{(J-1)}q_1, A^{(J-1)}q_2) + A(x^J)B^{(J-1)}(q_1, q_2). \tag{7}$$

$$\begin{aligned} C^{(J)}(q_1, q_2, q_3) = \quad & C(x^J)(A^{(J-1)}q_1, A^{(J-1)}q_2, A^{(J-1)}q_3)+ \\ & B(x^J)(B^{(J-1)}(q_1, q_2), A^{(J-1)}q_3)^*+ \\ & A(x^J)(C^{(J-1)}(q_1, q_2, q_3)), \end{aligned} \tag{8}$$

where * means that all combinatorially different terms have to be included, i.e.

$$\begin{aligned} B(x^J)(B^{(J-1)}(q_1, q_2), A^{(J-1)}q_3)^* \quad = \quad & B(x^J)(B^{(J-1)}(q_1, q_2), A^{(J-1)}q_3) + \\ & B(x^J)(B^{(J-1)}(q_1, q_3), A^{(J-1)}q_2) + \\ & B(x^J)(B^{(J-1)}(q_2, q_3), A^{(J-1)}q_1). \end{aligned}$$

For $D^{(J)}$ we get

$$\begin{aligned} D^{(J)}(q_1, q_2, q_3, q_4) = \quad & D(x^J)(A^{(J-1)}q_1, A^{(J-1)}q_2, A^{(J-1)}q_3, A^{(J-1)}q_4)+ \\ & C(x^J)(B^{(J-1)}(q_1, q_2), A^{(J-1)}q_3, A^{(J-1)}q_4)^*+ \\ & B(x^J)(B^{(J-1)}(q_1, q_2), B^{(J-1)}(q_3, q_4))^*+ \\ & B(x^J)(C^{(J-1)}(q_1, q_2, q_3)), A^{(J-1)}q_4)^*+ \\ & A(x^J)D^{(J-1)}(q_1, q_2, q_3, q_4). \end{aligned} \tag{9}$$

Finally, for $E^{(J)}$ holds

$$\begin{aligned} E^{(J)}(q_1, q_2, q_3, q_4, q_5) = \quad & E(x^J)(A^{(J-1)}q_1, A^{(J-1)}q_2, A^{(J-1)}q_3, A^{(J-1)}q_4, A^{(J-1)}q_5)+ \\ & D(x^J)(B^{(J-1)}(q_1, q_2), A^{(J-1)}q_3, A^{(J-1)}q_4, A^{(J-1)}q_5)^*+ \\ & C(x^J)(B^{(J-1)}(q_1, q_2), B^{(J-1)}(q_3, q_4), A^{(J-1)}q_5)^*+ \\ & C(x^J)(C^{(J-1)}(q_1, q_2, q_3), A^{(J-1)}q_4, A^{(J-1)}q_5)^*+ \\ & B(x^J)(C^{(J-1)}(q_1, q_2, q_3), B^{(J-1)}(q_4, q_5))^*+ \\ & B(x^J)(D^{(J-1)}(q_1, q_2, q_3, q_4))(A^{(J-1)}q_5)^*+ \\ & A(x^J)(E^{(J-1)}(q_1, q_2, q_3, q_4, q_5)). \end{aligned} \tag{10}$$

A drawback of using these recursive formulas is the nonlinear growth rate of the time complexity when the iteration number J increases. To make it clear, we use e_1, e_2, e_3, e_4 and e_5 to indicate the complexity of computation of the multilinear functions, i.e.

$$\begin{aligned} e_1 &= e(Aq) \\ e_2 &= e(B(q_1, q_2)) \\ e_3 &= e(C(q_1, q_2, q_3)) \\ e_4 &= e(D(q_1, q_2, q_3, q_4)) \\ e_5 &= e(E(q_1, q_2, q_3, q_4, q_5)). \end{aligned}$$

Then complexity for the multilinear forms up to the fifth order using the recursive formulas (6), (7), (8), (9) and (10), respectively, is given by:

$$\begin{aligned} e(A^{(J)}q) &= Je_1 \\ e(B^{(J)}(q_1, q_2)) &= (J-1)(J+1)e_1 + Je_2 = J^2e_1 + Je_2 + \text{ lower order terms} \\ e(C^{(J)}(q_1, q_2, q_3)) &= J^3e_1 + J^2e_2 + Je_3 + \text{ lower order terms} \\ e(D^{(J)}(q_1, q_2, q_3, q_4)) &= J^4e_1 + J^3e_2 + J^2e_3 + Je_4 + \text{ lower order terms} \\ e(E^{(J)}(q_1, q_2, q_3, q_4, q_5)) &= J^5e_1 + J^4e_2 + J^3e_3 + J^2e_4 + Je_5 + \text{ lower order terms.} \end{aligned}$$

4. Test cases

In this section, we present test cases to compare the accuracy and speed of AD and SD in the computation of the normal form coefficients of bifurcations of cycles in the software CL_MATCONTM, see [7, 11].

For convenience we mention the normal forms of the map at bifurcation points of type *flip* (PD) and *fold+flip* (LPPD) in which we will compute the critical normal form coefficients. The normal forms of all codimension-2 bifurcations of fixed points with at most two critical eigenvalues are given in [14].

4.1. Normal form coefficients of PD *and* LPPD *bifurcation points*

At a PD bifurcation point of J-cycles of f, the Jacobian matrix of $f^{(J)}$ in (1) has a simple eigenvalue $\lambda_1 = -1$ and no other eigenvalues on the unit circle. The restriction of (1) to a one-dimensional center manifold at the critical parameter value can be transformed to the normal form

$$w \mapsto -w + \frac{1}{6}bw^3 + O(w^4), w \in R^1 \tag{11}$$

where α is a control parameter. The normal form coefficient (NFC) b is given by:

$$b = \tfrac{1}{6}\langle p, C^{(J)}(q,q,q) + 3B^{(J)}(q, (I_n - A^{(J)})^{-1}B^{(J)}(q,q))\rangle, \tag{12}$$

where I_n is the unit $n \times n$ matrix, $A^{(J)}q = -q$, $[A^{(J)}]^T p = -p$ and $\langle q, q\rangle = \langle p, q\rangle = 1$. Here $\langle\ ,\ \rangle$ denotes the inner product.

A LPPD bifurcation is characterized by two simple eigenvalues on the unit circle, one $+1$ and one -1. Near a LPPD bifurcation, the restriction of (1) to the parameter-dependent center

J	bifurcation point
3	(0.375974, −0.627941, 1.335343)
6	(0.349755, −0.948447, 1.436871)
12	(0.336832, −0.881886, 1.463676)
24	(0.330787, −0.857379, 1.469354)
48	(0.329173, −0.852556, 1.470561)
96	(0.3391975, −0.925268, 1.470819)
192	(0.339223, −0.925734, 1.470874)

Table 1: PD bifurcation points of iterates of M_K. The left column contains the iteration numbers.

manifold is smoothly equivalent to the normal form

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \mapsto \begin{pmatrix} \beta_1 + (1+\beta_2)w_1 + a(\beta)w_1^2 + b(\beta)w_2^2 + c_1(\beta)w_1^3 + c_2(\beta)w_1w_2^2 \\ -w_2 + e(\beta)w_1w_2 + c_3(\beta)w_1^2w_2 + c_4(\beta)w_2^3 \end{pmatrix} + \quad O(\|w\|^4), \quad w \in \mathbb{R}^2. \tag{13}$$

Here, $\beta = (\beta_1, \beta_2)$ is an unfolding vector which vanishes at the bifurcation point, and the functions $a(\beta)$, etc., are defined in [14]. The normal form coefficients are their values at $\beta = 0$, which are the only values of these functions that are of concern in this paper.

4.2. Test case 1

We consider a 2-dimensional difference equation with 3 parameters:

$$M_K : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} Sx - y - (\epsilon y^2 + x^2) \\ Lx - (y^2 + x^2)/5 \end{pmatrix}. \tag{14}$$

where ϵ, L and S are parameters (unpublished PhD thesis of A. Yu. Kuznetsova, Saratov university).

$M_K^{(3)}$ has a fixed point at $(x^*, y^*) = (0.37588802742303, -0.62783638474655)$ when the parameter values are given by $(\epsilon, L, S) = (1, 1.3353, -0.799600)$. Continuation of fixed points of the third iterate, with L free and keeping ϵ, S fixed, leads to a supercritical flip bifurcation point when $L = 1.335343$. The map $M_K^{(3)}$ has a cascade of flip points that can be computed by switching to the new branches of double period at the PD points. We compute the PD points of the order 3, 6, 12, 24, 48, 96, 192 and then compute their successive NFCs. The coordinates of the PD points (x, y, L) are given in Table 1. The computed values and elapsed time are given in Table 2 and depicted in Figure 1, left. As expected, the computed values are equal to machine precision.

In the case of AD, the elapsed time grows linearly with the iteration number J (i.e., the slope is approximately 1 in the log-log plot in Figure 1) whereas in the case of SD the elapsed time grows as a higher power of J. However, for the iterates of low order 3,6,12, SD is still faster than AD, with crossover at around iteration 24.

Tests were also done with centered finite differences. As expected, the accuracy with FD was very poor and in several cases not even the sign of the coefficient was computed correctly.

We continue the flip bifurcation curves of the iterates of M_K starting from the computed PD points with L and S as bifurcation parameters, keeping ϵ fixed, and detect codimension-2 bifurcation points (LPPD). Their coordinates (x, y, L, S) are given in Table 3. The computed

J	b	t_{SD}	t_{AD}
3	5.662603e+3	0.041	0.074
6	8.753606e+1	0.047	0.086
12	5.807277e+2	0.080	0.135
24	2.080773e+4	0.225	0.222
48	6.023199e+5	1.13	0.401
96	4.881335e+6	8.00	0.764
192	1.969764e+8	66.6	2.90

Table 2: Computed NFC b and elapsed time, in seconds, in a cascade of PD points. The values of b computed by SD and AD are identical to within round-off.

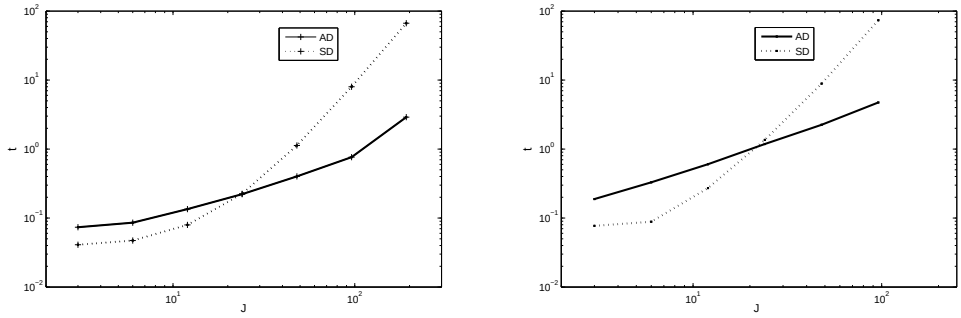

Figure 1: Elapsed times, in seconds, as a function of the iteration number, for normal form computations using SD and AD. Left: for the PD points. Right: for the LPPD points.

NFCs $a(0)$ and $b(0)$ and elapsed time are given in Table 4 and depicted in Figure 1, right.

It is clear from Figure 1 that the growth of elapsed time in the computation of NFCs of the LPPD points is apparently linear for AD and far more rapid for SD, again with crossover at around iteration 24.

4.3. Test case 2

The origins of the present map (15) can be found in [1, 5, 6]. It is two-dimensional with seven parameters described in [17], as follows.

$$M_{CS} : \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mapsto \begin{pmatrix} F.e^{-\beta_1 x_2} x_2 + (1-\mu_1) e^{-\beta_2 x_2} x_1 \\ P.e^{-\beta_3 x_2} x_1 + (1-\mu_2) x_2 \end{pmatrix} \tag{15}$$

J	bifurcation point
3	(−0.089643, −0.664722, 1.249136, −1.474424)
6	(0.297387, −0.923383, 1.468821, −0.746244)
12	(0.310548, −0.873006, 1.504478, −0.731940)
24	(0.316667, −0.855444, 1.510727, −0.729198)
48	(0.317656, −0.850743, 1.512561, −0.728413)
96	(0.296337, −0.907914, 1.512765, −0.728324)

Table 3: LPPD bifurcation points (x, y, L, S) of iterates of M_K. The left column contains the iteration numbers.

J	$a(0)$	$b(0)$	t_{SD}	t_{AD}
3	2.715094e+0	1.173625e+0	0.077	0.186
6	3.814972e+0	1.275108e+1	0.088	0.330
12	1.783747e+0	–7.907552e+1	0.272	0.604
24	7.404406e–1	2.184766e+3	1.36	1.18
48	3.603919e–1	–8.539430e+4	8.86	2.25
96	1.337659e–1	8.791200e+5	73.7	4.73

Table 4: Computed coefficients and elapsed time, in seconds, in the computation of normal form coefficients of iterates of M_K at LPPD bifurcation points. The results computed by SD and AD are identical to within round-off.

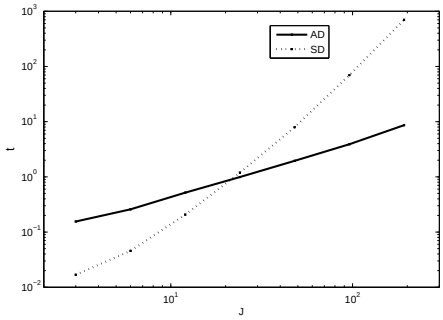

Figure 2: Elapsed time, in seconds, in the computation of normal forms of PD points, using SD and AD for the map M_{CS}.

where x_1 and x_2 are the immature and mature parts of the cod stock (at some time t) respectively and $F, P, \beta_1, \beta_2, \beta_3, \mu_1$ and μ_2 are dynamics parameters. Overall dynamical behavior of M_{CS} was studied in [12] and [17].
$M_{CS}^{(3)}$ has a fixed point at $X^* = (x_1^*, x_2^*) = (26.16934, 3.04173)$ for $F = 399.5861, \mu_1 = 0.5, \mu_2 = 0.444715, \beta_1 = \beta_2 = \beta_3 = 1$ and $P = 0.5$. We continue the fixed point of $M_{CS}^{(3)}$ starting from X^* with free parameter μ_2 and find a cascade of period doubling points that can be computed by switching to new branches of period 6, 12, 24, etc. We compute the NFC of the PD points of iterates 3, 6, 12, 24, 48, 96, 192 and compare the speed of SD and AD. The results are given in Figure 2. As in test case 1, the elapsed time grows apparently linearly for AD and much faster for SD, with crossover at around iteration 24.

5. Conclusions

We have proved that AD of functions of one variable can effectively be used to compute higher order directional derivatives of functions of several variables. We have applied this to the computation of normal form coefficients of bifurcation points of iterates of maps. We have shown that for high iteration numbers this method outperforms the use of symbolic derivatives, at least in the MATLAB environment. For low iteration numbers AD is slower but could be useful when no symbolic toolbox is available. In our applications we need derivatives up to order five, higher than any finite difference method could handle.

In any environment we expect that the AD cost would grow linearly with the number of iterations while the SD cost would grow roughly like the iteration number raised to the power of the highest derivative used.

References

[1] H. Caswell, Matrix Population Models: Construction, Analysis, and Interpretation, second ed., Sinauer Associates, Sunderland MA, 2001.

[2] T. F. Coleman, A. Verma, ADMAT: An automatic differentiation toolbox for MATLAB, Tech. rep., Computer Science Department, Cornell University, 1998.

[3] T. F. Coleman, A. Verma, The efficient computation of sparse Jacobian matrices using automatic differentiation, SIAM J. Sci. Comput. 19(4) (1998) 1210–1233.

[4] T. F. Coleman, A. Verma, ADMIT-1: Automatic differentiation and MATLAB interface toolbox, ACM Trans. Math. Softw. 26(1) (2000) 150 – 175.

[5] J.M. Cushing, R.F. Costantino, B. Dennis, R.A. Deshamias, A.M. Henson, Nonlinear population dynamics: models, experiments and data, J. Theor. Biol. 194 (1998) 1–9.

[6] B. Dennis, R.A. Deshamais, J.M. Cushing, R.F. Costantino, Transition in population dynamics: equilibria to periodic cycles to aperiodic cycles, J. Anim. Ecol. 6b (1997) 704–729.

[7] A. Dhooge, W. Govaerts, Yu.A. Kuznetsov, W. Mestrom, A. Riet, CL_MATCONT: A continuation toolbox in Matlab, `http://sourceforge.net/projects/matcont`, 2004.

[8] S.A. Forth, M. Edvall, User Guide for MAD: a MATLAB Automatic Differentiation Toolbox, Version 1.4, Engineering Systems Dept, Cranfield University, 2007.

[9] A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, Philadelphia, 2000.

[10] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, second ed., SIAM, Philadelphia, 2008.

[11] W. Govaerts, R. Khoshsiar Ghaziani, Yu. A. Kuznetsov, H.G.E. Meijer, Numerical methods for two-parameter local bifurcation analysis of maps, SIAM J. Sci. Comput. 29(6) (2007) 2644–2667.

[12] W. Govaerts, R. Khoshsiar Ghaziani, Numerical bifurcation analysis of a nonlinear stage structured cannibalism population model, J. Diff. Eqns. Appl. 12(10) (2006) 1069–1085.

[13] Yu.A. Kuznetsov, Elements of Applied Bifurcation Theory, third edition, Springer-Verlag, New York, 2004.

[14] Yu.A. Kuznetsov, H.G.E. Meijer, Numerical normal forms for codim 2 bifurcations of maps with at most two critical eigenvalues, SIAM J. Sci. Comput. 26(6) (2005) 1932–1954.

[15] L.C. Rich, D.R. Hill, Automatic differentiation in MATLAB, App. Num. Math. 9 (1992) 33–43.

[16] A. Verma, Structured automatic differentiation, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[17] A. Wikan, A. Eide, An analysis of a nonlinear stage-structured cannibalism model with application to the Northeast Arctic cod stock, Bull. Math. Biol. 66 (2004) 1685–1704.